JFP **23** (3): 229–248, 2013. © Cambridge University Press 2013
doi:10.1017/S0956796813000063

A library for polymorphic dynamic typing

WOUTER SWIERSTRA
Utrecht University, Viale R. Elena, 324, 00185 Rome, Italy

THOMAS VAN NOORT
Radboud University Nijmegen, Comeniuslaan 4, 6525 HP Nijmegen, Netherlands
(*e-mail:* `w.s.swierstra@uu.nl` and `thomas@cs.ru.nl`)

Abstract

This paper presents a library for programming with polymorphic dynamic types in the dependently typed programming language Agda. The resulting library allows dynamically typed values with a *polymorphic type* to be instantiated to a less general (possibly monomorphic) type without compromising type soundness.

1 Introduction

There are situations where the types of the values that a program manipulates are not known during compilation. This is typically the case when data, or even parts of the program itself, are obtained by interacting with the 'outside' world: when values are exchanged between applications by deserialization from disk, input is provided by a user, or part of a program is obtained over a network connection.

Modern statically typed functional languages, such as Clean (Van Eekelen *et al.*, 1990), Haskell (Peyton Jones, 2003), and OCaml (Leroy *et al.*, 2011), all support some form of *dynamic typing* that allows programmers to defer type checking until runtime. These languages define a special type for dynamically typed values. We will abbreviate such dynamically typed values to *dynamic value* or just *dynamic*. A dynamically typed value consists of a value packaged together with a representation of that value's type. A programmer may attempt to coerce a dynamic value to a value with a statically known type and such coercions may fail at run time. If such a coercion succeeds, however, the type soundness of the rest of the program should not be compromised.

There are several differences between the forms of dynamic typing that are supported by Clean and Haskell. In Haskell, dynamic typing is supported by means of a library, built using several GHC language extensions (Lämmel & Peyton Jones, 2003). The Haskell library for dynamic types provides a function toDyn that wraps a value in a dynamic:

```
incDyn :: Dynamic
incDyn  = let inc :: Int → Int
              inc x = x + 1
          in toDyn inc
```

In Haskell, there are some limitations on which values can be wrapped in a dynamic. In particular, Haskell only allows *monomorphic* values to be stored in a dynamic. In order to wrap a *polymorphic* value in a dynamic, we need to instantiate its type explicitly. For example, the following code packages a monomorphic identity function, instantiated to work on integers, in a dynamic:

```
idDyn :: Dynamic
idDyn = let id :: forall a . a → a
            id x = x
        in toDyn (id :: Int → Int)
```

A value may be unwrapped using the library function fromDyn. The type to which the dynamic must be cast is inferred from the context:

```
idInt :: Maybe (Int → Int)
idInt = fromDyn idDyn
```

Now consider the following example:

```
fail :: Maybe (Bool → Bool)
fail = fromDyn idDyn
```

Although the value stored in the dynamic is the identity function, we had to instantiate its type explicitly to be Int → Int. Coercing the identity on integers to the identity on booleans fails, and hence the example above returns Nothing. This illustrates some of the limitations of Haskell's approach to dynamic typing.

Clean's support for dynamic typing, on the other hand, is built into the language definition. In contrast to Haskell, the Clean compiler allows *any* value (of a non-abstract type) to be stored in a dynamic, including polymorphically typed values. In Clean, any value can be wrapped in a dynamic by using the **dynamic** keyword:

```
idDyn :: Dynamic
idDyn = dynamic (λ x → x)
```

Then we can unwrap a dynamic by pattern matching and providing an explicit-type annotation:

```
id :: Maybe (A.a : a → a)
id = case idDyn of
       (f :: A.a : a → a) → Just f
       _                  → Nothing
```

In Clean, the notation A.a introduces a universal quantifier that binds the type variable a. The Clean type A.a : a → a would be written **forall** a . a → a in Haskell. This example shows how to cast dynamic values to a *polymorphic* type in Clean.

It is important to observe that the required type does not need to be structurally equal to the type found in the dynamic: it is allowed to be more specific than the type of the dynamic value. For example, suppose we require the result to be a function of the type Int → Int:

```
idInt :: Maybe (Int → Int)
idInt = case idDyn of
          (f :: Int → Int) → Just f
          _                → Nothing
```

Here the compiler checks that the desired type is an instance of the type of the value in the dynamic. When this succeeds, the value is implicitly coerced to the required type. Being able to store an arbitrary, polymorphic value as a dynamic value turns out to have several important applications (Plasmeijer & Van Weelden, 2005; Plasmeijer *et al.*, 2011).

This paper describes a *library* for dynamic typing capable of handling *polymorphic* values, thereby combining the advantages of both Haskell and Clean's dynamic typing mechanisms.

Throughout this paper we will use Agda (Norell, 2007, 2008), a programming language with dependent types, to carry out this investigation. This may seem like a peculiar choice: why introduce a third programming language? As we shall see, defining the desired library for dynamically typed programming requires some 'programming with types'. Although we believe that it is possible to define such a library in Haskell (and indeed others have proposed to do so), we have chosen to work in a language most suited for such a development. In the future, we hope to investigate how our library may be backported to Haskell. This does limit the practical applications of our library: despite recent progress (Brady, 2011), it is still cumbersome to compile Agda code and to interface with Haskell.

One further advantage of working in Agda is that we cannot cut corners. The library we define does not use compiler primitives, it does not extend the language, and it does not require postulates or assumptions. As a result, the code we present is not only a library for programming with dynamic types but may also be seen as a mathematical specification of Clean's dynamic types, together with a mechanized proof of type soundness.

2 Monomorphic dynamics

In this section we use Agda to define a small library for monomorphic dynamic typing. In later sections, we will show how this can be extended to handle polymorphism. Note that the code we present relies on a small Agda prelude that defines heterogeneous equality, natural numbers, and several familiar Haskell types.

The central concept that underlies programming with generics and dynamics in a dependently typed language is that of a *universe* (Martin-Löf, 1984; Altenkirch & McBride, 2003; Oury & Swierstra, 2008). A universe consists of a data type U encoding some collection of types and a 'decoding' function el : U → Set that maps every code to the type it represents. To make this more concrete, consider the following universe definition:

```
data U : Set where
  NAT  : U
  PAIR : U → U → U
  _⇒_ : U → U → U
el : U → Set
el NAT = Nat
el (PAIR u₁ u₂) = Pair (el u₁) (el u₂)
el (u₁ ⇒ u₂)    = el u₁ → el u₂
```

This defines a data type U with three constructors and a function mapping every element of U to the type it represents. For example, the constructor NAT is used to represent natural numbers. The el function maps NAT to Nat, the inductively defined type of natural numbers. This universe is also closed under two type constructors: pairs and function spaces. Although we could also add other types or type constructors, such as the unit type, the empty type, co-products, fixed points, and so forth, we will restrict ourselves to these two type constructors. Crucially, the constructions we present do not rely on the covariance or contravariance of the type constructors in the universe of discourse.

A dynamic value consists of an element of this universe, paired with a value of the type that it represents:

```
data Dynamic : Set where
  Dyn : (u : U) → el u → Dynamic
```

Next, we need to define a cast function with the following type:

```
cast : (u : U) → Dynamic → Maybe (el u)
```

Intuitively, a call to cast u dyn should check if the value stored in dyn has type el u. If so, it should successfully return the value stored in the dynamic; otherwise, it should fail and return Nothing.

To check whether or not two inhabitants of U are structurally equal, we need to define the following function:

$$\mathsf{decEqU} : (u_1\ u_2 : \mathsf{U}) \to \mathsf{Either}\ (u_1 \equiv u_2)\ (u_1 \not\equiv u_2)$$

Here the statement $u_1 \equiv u_2$ refers to the usual notion of propositional equality between u_1 and u_2; the statement $u_1 \not\equiv u_2$ is the negation of this equality. The definition proceeds by simultaneous induction on both u_1 and u_2. As it is entirely straightforward, we have omitted it from this paper.

Using this auxiliary function, we can now define the cast function as follows:

```
cast : (u₁ : U) → Dynamic → Maybe (el u₁)
cast u₁ (Dyn u₂ x) with decEqU u₁ u₂
cast u₁ (Dyn ⌊u₁⌋ x) | Inl Refl = Just x
cast u₁ (Dyn u₂ x)   | _        = Nothing
```

The cast function decides whether or not the argument u_1 is equal to the type of the value stored in the dynamic using Agda's **with** construct (McBride & McKinna, 2004; Norell, 2007). If this is the case, we pattern match on the Refl constructor, from which we learn that the first component of the dynamic must be equal to u_1. In Agda, this information is recorded by a forced pattern, $\lfloor u_1 \rfloor$. In that case, we return the value stored in the dynamic. Otherwise the cast fails. Chapter 2 of Norell's thesis (2007) gives a more complete description of both forced patterns and the **with** construct.

3 Polymorphic dynamic typing

The universe we saw previously could only be used to represent a fairly small collection of monomorphic types. In this section we will show how to extend it with type variables. We represent type variables as De Bruijn indices using the datatype Fin n.

```
data Fin : Nat → Set where
  Fz : Fin (Succ n)
  Fs : Fin n → Fin (Succ n)
```

For any natural number n, the type Fin n has n distinct inhabitants. Note that in the typeset code presented in this paper, any unbound variables in type signatures are implicitly universally quantified, as is the convention in Haskell (Peyton Jones, 2003) and Epigram (McBride & McKinna, 2004). When we wish to be more explicit about implicit arguments, we will adhere to Agda's notation of enclosing such arguments in curly braces.

We now extend the universe from the previous section with a new constructor for type variables:

```
data U (n : Nat) : Set where
  NAT  : U n
  PAIR : U n → U n → U n
  _⇒_ : U n → U n → U n
  VAR  : Fin n → U n
```

The universe U is parametrized by a natural number, indicating the number of variables that type codes may use. Furthermore, we add a new constructor, VAR, for type variables. We will refer to inhabitants of U n as (codes for) *monotypes*.

Finally, we introduce a new data type V that wraps universal quantifiers around any monotype.

```
data V : Set where
  FORALL : {n : Nat} → U n → V
```

You may want to think of the FORALL constructor as wrapping n universal quantifiers around its argument monotype, ensuring that it is closed. We will refer to inhabitants of V as (codes for) *polytypes*.

Using the universes U and V, we can now represent the type of the polymorphic identity function as follows:

```
idType : V
idType = FORALL {Succ Zero} (VAR Fz ⇒ VAR Fz)
```

We have some degree of freedom about how many quantifiers to use. If we had written FORALL {Succ (Succ (Succ Zero))} (VAR Fz ⇒ VAR Fz), this would correspond to the Haskell type **forall** a b c . a → a.

Interpretation. Although we have defined the data types necessary to represent polymorphic types, we still need to define the interpretation functions mapping U and V to Set. Before we can do so, we need to define one auxiliary notion: type environments.

```
data Env : Nat → Set where
  Nil  : Env Zero
  Cons : U Zero → Env n → Env (Succ n)
```

An environment Env n consists of a list of exactly n closed monotypes. It is straightforward to define a function, findInEnv, that given an index and an environment, returns the monotype stored in the environment at that index:

```
findInEnv : Fin n → Env n → U Zero
findInEnv Fz      (Cons u _)   = u
findInEnv (Fs i) (Cons _ env)  = findInEnv i env
```

In the base case for Fz we return the first entry. In the case for Fs i, we make a recursive call on the index i and the tail of the environment.

Given a type environment, we can *close* any monotype, replacing any type variables by the closed monotypes to which they refer:

```
close : U n → Env n → U Zero
close NAT             _    = NAT
close (PAIR u₁ u₂) env   = PAIR (close u₁ env) (close u₂ env)
close (u₁ ⇒ u₂)    env   = close u₁ env ⇒ close u₂ env
close (VAR i)       env   = findInEnv i env
```

We can now define the interpretation of closed monotypes as follows:

```
elClosed : U Zero → Set
elClosed NAT             = Nat
elClosed (PAIR u₁ u₂)  = Pair (elClosed u₁) (elClosed u₂)
elClosed (u₁ ⇒ u₂)     = elClosed u₁ → elClosed u₂
elClosed (VAR ())
```

The elClosed function maps any *closed* monotype to the type it represents: the codes for natural numbers, pairs, and functions map to their respective types. The case for variables is ruled out, as we know that the monotype is closed.

To interpret an arbitrary monotype that may still contain variables, the elU function requires an additional type environment. It first closes the monotype, essentially substituting closed types for any variables. By calling elClosed we can then produce the desired type,

```
elU : U n → Env n → Set
elU u env = elClosed (close u env)
```

This may seem a bit clumsy: why not define elU directly by induction on the first argument? If you try to do so, there is a slight problem in the case branch for variables. The case for variables would consult the environment and then recursively call elU:

```
elU (VAR i) env = elU (findInEnv i env) Nil
```

Agda's termination checker is not able to see that this definition terminates – the recursive call is not on a structurally smaller subterm of the first argument, but on some arbitrary monotype stored in the environment. Although the monotype stored in the environment is closed, Agda's termination checker is not convinced that this branch will always terminate. Indeed, if we were to store monotypes in U n, for arbitrary n, in the environment this need not be the case. With the explicit stratification described above, Agda's termination checker happily accepts our definitions.

We can now define the interpretation of polytypes as follows:

```
elV : V → Set
elV (FORALL {n} u) = forall {env : Env n} → elU u env
```

The interpretation maps the FORALL constructor to an implicit universal quantification over an environment argument, and calls elU with this environment.

Before we move on to dynamics, there is one more design choice to point out. The environment contains closed monotypes, rather than polytypes or types in Set. This is not strictly necessary: the development we present below works if we allow the environment to store arbitrary types in Set. Doing so requires a move from Set to Set_1 in a handful of definitions. To keep the types in this presentation small, we limit ourselves to environments storing monotypes in this paper.

Using our new universes for monotypes and polytypes, we can now redefine the datatype Dyn to use polytypes:

```
data Dynamic : Set where
  Dyn : (v : V) → elV v → Dynamic
```

In contrast to the previous section, we can now wrap polymorphic values in a dynamic:

```
idDyn : Dynamic
idDyn = Dyn idType (λ x → x)
```

As a first approximation, we can redefine the cast function we saw previously to handle polytypes:

```
cast : (v₁ : V) → Dynamic → Maybe (elV v₁)
cast v₁ (Dyn v₂ x) with decEqV v₁ v₂
cast v₁ (Dyn ⌊v₁⌋ x) | Inl Refl = Just x
...                  | _        = Nothing
```

The only difference with the previous version of the cast function in Section 2 is that we now check whether or not two codes for polytypes, that is elements of V, are equal or not. The previous version of the cast function dealt with a simpler type universe that could only describe monomorphic types. Even though the universe V can describe polymorphic types, we still only check whether the two types involved are structurally equal. This check is done using the decEqV function, which itself uses the decEqU function from the previous section. Its definition is straightforward but not listed here.

This definition of cast does not quite give us the behaviour we would like to have. For example, consider the idDyn dynamic we defined above. When we try to cast it to the type of the polymorphic identity function, this will succeed:

```
success : cast idType idDyn ≡ Just (λ x → x)
success = Refl
```

Should we try to cast it to, say, the identity function on natural numbers, this will fail:

```
fail : cast (FORALL {Zero} (NAT ⇒ NAT)) idDyn ≡ Nothing
fail = Refl
```

The reason for this lies in the definition of the cast function: a cast will only succeed if the two types are structurally identical. Clearly, this is too strict a requirement. In the coming sections we will develop an alternative version of the cast function that instantiates the type of polymorphic dynamics when necessary.

4 Instantiation

What is the problem we need to address? Given two monotypes, u_1 and u_2, we need to find a substitution σ such that applying σ to u_2 is equal to u_1. When this is the case, we will say that σ instantiates u_2 to u_1. Before we can define an algorithm that addresses this problem, we need to define several functions to create and manipulate substitutions.

Substitutions. We begin by defining a data type to represent substitutions:

```
data PartSubst (m : Nat) : Nat → Set where
  Nil : PartSubst m Zero
  Cons : Maybe (U m) → PartSubst m n → PartSubst m (Succ n)
```

A value of type PartSubst m n consists of n values of type Maybe (U m). Such a value describes a *partial* substitution that instantiates *some* variables in U n to a type U m. If substitution has a Nothing at the i-th position, we have not yet encountered any constraints on how to instantiate VAR i; applying the substitution would leave the variable VAR i untouched. If the substitution does have a monotype at the i-th position, the substitution will replace variable VAR i by that monotype. This is an important distinction to make: a Nothing at the i-th position means no constraint, whereas VAR i means that this variable must remain unconstrained.

Our aim is to define a function, check, that finds an instantiating substitution. This function will be defined using an accumulating parameter: starting with the empty substitution, we will traverse both types simultaneously, collecting constraints on how type variables must be instantiated. This motivates the need for considering *partial* substitutions: during this traversal the substitution we have may not be complete yet. Instead of applying intermediate substitutions immediately during the traversal, we choose to work with partial substitutions to keep our check function structurally recursive.

The empty substitution is easily constructed by performing induction on the length of the substitution and inserting Nothing values at every position:

```
empty : {m : Nat} → PartSubst m n
empty {Zero}    = Nil
empty {Succ n}  = Cons Nothing empty
```

The empty substitution will be the initial accumulating parameter check function.

Just as we defined findInEnv, we can define findInPartSubst that looks up the type associated with a given variable:

```
findInPartSubst : Fin n → PartSubst m n → Maybe (U m)
findInPartSubst Fz     (Cons x _)      = x
findInPartSubst (Fs i) (Cons _ subst)  = findInPartSubst i subst
```

The type of findInPartSubst is a little more complex than that of findInEnv because substitutions may store types with uninstantiated variables, i.e. U n for some number n, whereas environments may only store closed types, i.e. U Zero. This definition illustrates the need for decoupling the length of the substitution n from the type of the values in the substitution. In the recursive call to findInPartSubst the length of the remaining substitution decreases, but the type returned stays the same.

The drawback of using partial substitutions is that the apply function is now also partial. The function apply traverses the argument monotype looking for variables. Once a variable is encountered, the apply function consults the substitution to try to find the monotype to which the substitution maps this variable.

```
apply : PartSubst m n → U n → Maybe (U m)
apply _      NAT             = Just NAT
apply subst (PAIR u₁ u₂)    = Just PAIR ⊛ apply subst u₁ ⊛ apply subst u₂
apply subst (u₁ ⇒ u₂)       = Just _⇒_ ⊛ apply subst u₁ ⊛ apply subst u₂
apply subst (VAR i)         = findInPartSubst i subst
```

This definition uses the applicative ⊛ operator (McBride & Paterson, 2008) to combine the results of the recursive calls:

```
_⊛_ : Maybe (a → b) → Maybe a → Maybe b
```

The last operation we will need on substitutions is the extend function defined below. The application extend i u subst extends the substitution subst so that the variable i will now be mapped to u. If subst already maps the variable i to a different monotype, the call to extend will fail.

```
extend : Fin n → U m → PartSubst m n → Maybe (PartSubst m n)
extend Fz     u (Cons Nothing subst)    = Just (Cons (Just u) subst)
extend Fz     u (Cons (Just u') _) with decEqU u u'
extend Fz     u (Cons (Just ⌊u⌋) subst) | Inl Refl = Just (Cons (Just u) subst)
extend Fz     u (Cons (Just u') _)      | Inr _    = Nothing
extend (Fs i) u (Cons mu subst)         = Just (Cons mu) ⊛ extend i u subst
```

The definition of extend does what you would expect: It traverses the substitution until it hits the i-th position. In the first branch, there is no monotype at the i-th position and the function returns a new substitution in the obvious fashion. If there is already a monotype at the i-th position, we check whether that monotype is equal to the argument monotype u. If so, we leave the substitution unchanged; if not, the extend function fails and returns Nothing. The final branch simply continues traversing the substitution.

Instantiation check. Now that we have defined substitutions, we continue by defining the actual instantiation check. Given two types, u_1 and u_2, the check function determines if the first argument is an instance of the second argument, and if so, returns the instantiating substitution. The check function is defined using an accumulating parameter that is initially the empty substitution in Figure 1.

The checkAcc simultaneously traverses both its type arguments, threading the accumulating substitution through the recursive calls. The resultsof the recursive calls are

```
check : U n → U m → Maybe (PartSubst n m)
check u₁ u₂ = checkAcc u₁ u₂ empty

checkAcc : U n → U m → PartSubst n m → Maybe (PartSubst n m)
checkAcc NAT            NAT            subst = Just subst
checkAcc (PAIR u₁ u₂) (PAIR w₁ w₂) subst = checkAcc u₁ w₁ subst >>=
                                              checkAcc u₂ w₂
checkAcc (u₁ ⇒ u₂)    (w₁ ⇒ w₂)    subst = checkAcc u₁ w₁ subst >>=
                                              checkAcc u₂ w₂
checkAcc u              (VAR i)        subst = extend i u subst
checkAcc _              _              _     = Nothing
```

Fig. 1. The instantiation check.

combined using the bind operation of the Maybe monad. If the two types differ at a non-variable position, we know there is no substitution that can equate them, and we return Nothing. The only interesting branch is the case where the second monotype is a variable. In that case, we attempt to extend the substitution so that the variable encountered will map to u.

Correctness. Before we use the check function to define a cast function, we need to establish a few correctness properties. In particular, we need to show that if check u_1 u_2 successfully produces a substitution σ, then apply σ u_2 $\equiv$ Just u_1. The cast function will use this equality to coerce its argument to the desired type.

This key property that we need to prove can be formulated in Agda as follows:

```
checkAccCorrect : (u₁ : U n) (u₂ : U m) →
  (subst subst' : PartSubst n m) →
  checkAcc u₁ u₂ subst ≡ Just subst' →
  apply subst' u₂ ≡ Just u₁
```

This property is a bit more general than you might expect. Taking *any* substitution subst as starting point for the accumulating parameter of checkAcc, we can show that the desired equality holds, provided our instantiation check succeeds. It may come as a surprise that any initial choice of substitution suffices. This only works because the instantiation check traverses u_2, updating the accumulating substitution. If there is any discrepancy between the information already present in the substitution and the information gathered during this traversal, the instantiation algorithm would have failed.

The proof of checkAccCorrect proceeds by induction on the monotypes u_1 and u_2. Rather than present in its full glory, we will outline the definitions and lemmas necessary. The off-diagonal cases of the checkAccCorrect lemma are easily discharged by the assumption that instantiation was successful. There are only three interesting cases: the branch for pairs, the branch for functions, and the branch for variables.

- In the first two cases, for pairs and functions, we traverse the constituent monotypes. To glue together the results, we need an additional lemma.

```
stability : (u₁ : U m) (u₂ : U n) (w₁ : U m) (w₂ : U n)
  (subst subst' subst'' : PartSubst m n) →
  checkAcc u₁ u₂ subst ≡ Just subst' →
  checkAcc w₁ w₂ subst' ≡ Just subst'' →
  apply subst'' u₂ ≡ apply subst' u₂
```

Roughly speaking, this property states that if the substitution subst' instantiates u_2 to u_1, and we extend subst' to some new substitution subst'', then subst'' also instantiates u_2 to u_1. The proof of the stability lemma is lengthy, but not conceptually difficult.

- The variable branch of the checkAccCorrect lemma is reasonably straightforward. The proof requires several auxiliary lemmas relating the findInPartSubst and extend functions. For example, the following property requires a short inductive proof:

```
extend i u subst ≡ Just subst' → findInPartSubst i subst' ≡ Just u
```

Instantiating values. The instantiation check, check u_1 u_2, tries to compute a substitution that instantiates the monotype u_2 to u_1. The question we still need to address is how to use this substitution to instantiate a *value* of the *polytype* FORALL u_2.

Recall that a value inhabiting elV (FORALL u) is a function taking an (implicit) environment env to a value of type elU u env. The only way to instantiate such a function is by *constructing* an *environment* that we can pass as an argument. The instantiate function that we will define below does just this.

In what follows, we will work with *total* substitutions, as opposed to the partial substitutions we have seen so far. Therefore, we begin by defining a new type Subst, representing total substitutions, as a dependent pair consisting of a partial substitution and a proof of its totality:

```
isTotal : PartSubst n m → Set
isTotal Nil                    = Unit
isTotal (Cons Nothing _)       = Empty
isTotal (Cons (Just _) subst)  = isTotal subst
data Subst (n m : Nat) : Set where
  _,_ : (subst : PartSubst n m) → isTotal subst → Subst n m
```

Using this notion of substitution, we can now define our instantiate function as follows:

```
instantiate : Env n → Subst n m → Env m
instantiate env (Nil, p)                 = Nil
instantiate env (Cons Nothing subst, ())
instantiate env (Cons (Just i) subst, p) =
  Cons (close i env) (instantiate env (subst, p))
```

The instantiate function traverses its argument substitution. Every type that occurs in the substitution is closed using the argument environment, to produce a new environment of the desired length. As the substitution is assumed to be *total*, we are free to discharge the case branch where no type is encountered. This definition also makes clear why we need the substitution to be *total*. If the substitution is not total, we would need to invent a closed monotype 'out of thin air' to produce an environment of the desired length.

We can prove the following characteristic property of instantiate:

```
instantiateCorrect : (u₁ : U n) (u₂ : U m) (subst : PartSubst n m)
  (env : Env n) → (p : isTotal subst) →
  apply subst u₂ ≡ Just u₁ →
  elU u₂ (instantiate env (subst, p)) ≡ elU u₁ env
```

This theorem states that given an instantiating substitution from u_2 to u_1, we can convert an element of u_1 by instantiating its environment. The proof, by induction on u_1 and u_2, is unsurprising: off-diagonal cases are discharged; pairs and functions require the application of two induction hypotheses; variables require an additional lemma about instantiate.

With all this machinery in place, we can now return to the original problem: how to cast a polymorphic dynamic?

5 Casting

Our aim is to define a cast function that is capable of instantiating polymorphic dynamic types. To do so, we can call our check function that compares two monotypes and tries to compute an instantiating partial substitution. We would like to compute an environment to pass to the polymorphic value stored in the dynamic using the instantiate function we have defined previously. Unfortunately, our instantiate function only works for *total* substitutions and the result of check function produces a *partial* substitution. We still have a bit more work to do.

Total substitutions. When does our check function not produce a *total* substitution? It traverses the second monotype argument and finds a type to assign to each type variable *in that monotype*. If that monotype does not contain all the type variables that have been quantified over, however, the check function will not find a type with which to instantiate that value. For instance, when constructing an instantiating substitution from the polytype **forall** a b . a → a to Nat → Nat, our check function does not produce a type with which to instantiate the type variable b. As a result, the substitution resulting from our instantiation check is not total.

There are different solutions to this problem. We could choose to instantiate type variables that do not matter with some arbitrary type such as Nat. This would require several proofs that it is safe to do so. While this solution does work, it yields a proof that is 'correct by coincidence' – it relies on the implicit assumption that certain type variables do not occur. This seems to defeat the whole purpose of working in a dependently typed language in the first place – we try to choose our types and function definition to rule out impossible or uninteresting cases. Therefore, we choose to make this assumption explicit in our definition of dynamically typed value.

We would like to enforce that types stored in a Dynamic do not contain spurious quantifiers. To do so, we start by defining the Elem relation that witnesses that a type variable occurs in a monotype,

```
data Elem (i : Fin n) : U n → Set where
  Here      : Elem i (VAR i)
  LeftPair  : Elem i l → Elem i (PAIR l r)
  RightPair : Elem i r → Elem i (PAIR l r)
  LeftFun   : Elem i l → Elem i (l ⇒ r)
  RightFun  : Elem i r → Elem i (l ⇒ r)
```

The base case states that the variable i occurs in the monotype VAR i. The other cases state that if a type variable occurs in any subtree of a monotype u, it also occurs in u.

We can now lift this Elem relation to hold for *all* values of type Fin n. To do so, we define the function allFin that lifts any predicate P on Fin n, to the proposition that states that P holds for every choice of Fin n

```
allFin : {n : Nat} → (Fin n → Set) → Set
allFin {Zero}   P = Unit
allFin {Succ y} P = Pair (P Fz) (allFin (λ n → P (Fs n)))
```

We call a monotype in U n *strong* if it contains all n distinct variables. (The term *strong* suggests that it has not been *weakened*). We can use the allFin function to define an isStrong predicate on monotypes

```
isStrong : U n → Set
isStrong u = allFin (λ i → Elem i u)
```

Using these definitions, we can now prove that if a partial substitution can be successfully applied to a strong monotype, this substitution must be total:

```
proveTotality : (u₁ : U n) (u₂ : U m) (subst : PartSubst n m) →
  isStrong u₂ →
  apply subst u₂ ≡ Just u₁ →
  isTotal subst
```

The proof uses an additional lemma, proved by induction over the Elem relation, that findInPartialSubst will successfully return a result for every variable in u_2.

Cast. We modify our Dynamic type to incorporate a new assumption:

```
data Dynamic : Set where
  Dyn : (u : U n) → isStrong u → elV (FORALL u) → Dynamic
```

We require all types stored in a Dynamic to be strong. This does add some burden to the users of our library as they need to write explicit proofs that a type is strong. We will return to this point shortly. For the moment, we proceed by finally completing our definition of cast in Figure 2.

The cast function starts by calling check in an attempt to find an instantiating substitution from u_2 to u_1. If this fails, the cast fails. If this succeeds, the cast succeeds, but we need to provide a value of type elV (FORALL u_1). To do so, we use the instantiate function to produce an environment, which we pass to the polymorphic value stored in the dynamic. As the instantiate function requires a total substitution as argument, we use the assumption that u_2 is strong to prove that the substitution resulting from our instantiation check is total.

```
cast : (u₁ : U m) → Dynamic → Maybe (elV (FORALL u₁))
cast u₁ (Dyn u₂ p x) with inspect (check u₁ u₂)
cast u₁ (Dyn u₂ p x) | Nothing by _ = Nothing
cast u₁ (Dyn u₂ p x) | Just subst by eq =
  Just (λ {env} → coerce (correct env) (x {instantiate env totalSubst}))
  where
  coerce : a ≡ b → a → b
  coerce Refl x = x
  substProp : apply subst u₂ ≡ Just u₁
  substProp = checkAccCorrect u₁ u₂ empty subst eq
  substTotality : isTotal subst
  substTotality = proveTotality u₁ u₂ subst p substProp
  totalSubst = (subst, substTotality)
  correct : (env : Env m) → elU u₂ (instantiate env totalSubst) ≡ elU u₁ env
  correct env = instantiateCorrect u₁ u₂ subst env substTotality substProp
```

Fig. 2. The final cast function.

The instantiation produces a value of type elU u_2 (instantiate env totalSubst) rather than the desired type elU u_1 env. Fortunately, we can prove that these two types are equal using our instantiateCorrect lemma, which in turn relies on the checkAccCorrect lemma. The coerce function uses this result to assign the desired type to the instantiated value. This cast function brings together all the definitions and lemmas that we have defined in the preceding pages.

There is one last subtlety in the definition of the cast function. We require an equality proof to use the checkAccCorrect lemma, stating that the call to check was successful. Despite having already pattern-matched on a Just constructor, we cannot provide the required proof. The usual workaround in Agda is to define the following auxiliary data type and function:

```
data Inspect {a : Set} (x : a) : Set where
  _by_ : (y : a) → x ≡ y → Inspect x
inspect : (x : a) → Inspect x
inspect x = x by Refl
```

Now by pattern matching on inspect (check u_1 u_2) rather than just check u_1 u_2, we may refer to the required equality proof when we need it in the remainder of the case branch.

Automation. This new version of our Dynamic type has a clear drawback: explicit proofs of strength must be constructed. For example, suppose we start by defining the type of the const function:

```
constType : U 2
constType = (VAR Fz) ⇒ ((VAR (Fs Fz)) ⇒ (VAR Fz))
```

To package the polymorphic const function as a dynamic value, we now need to provide an explicit proof object:

```
constDyn : Dynamic
constDyn = Dyn constType strength (\x y → x)
  where
  strength = (LeftFun Here, (RightFun (LeftFun Here), unit))
```

While the explicit type annotation u is bad enough, the strength proof that u contains each bound variable is fairly ugly. Fortunately, such proofs can be easily computed, as we will sketch here.

To compute such proofs, we start by defining a function that checks whether or not a certain variable occurs in a type. In contrast to the Elem data type, this function *computes* a boolean:

```
isElem : (i : Fin n) → U n → Bool
isElem i NAT            = False
isElem i (PAIR u1 u2)   = or (isElem i u1) (isElem i u2)
isElem i (u1 ⇒ u2)      = or (isElem i u1) (isElem i u2)
isElem i (VAR j)        = eqFin i j
```

In the usual fashion, we can turn any boolean into a proposition that is only inhabited when the boolean holds:

```
So : Bool → Set
So True  = Unit
So False = Empty
```

Using the allFin function defined previously, we can now give an alternative formulation for the predicate that checks that a monotype is strong:

```
isSoStrong : U n → Set
isSoStrong u = allFin (λ i → So (isElem i u))
```

This predicate has an important property: for any *closed* value u of type U n, when the type isSoStrong u is inhabited, it only consists of pairs of unit values. Furthermore, we can show that this alternative isSoStrong predicate implies the original isStrong predicate. We refer to this result as soundness:

```
soundness : (u : U n) → isSoStrong u → isStrong u
soundness u p = map (isElemSoundness u) p
  where
  map : {n : Nat} → ((i : Fin n) → P i → Q i) → allFin P → allFin Q
  map {Zero}   H _          = unit
  map {Succ k} H (p1, p2)  = (H Fz p1, map (λ i → H (Fs i)) p2)
  isElemSoundness : (u : U n) (i : Fin n) → So (isElem i u) → Elem i u
```

We have omitted the proof of isElemSoundness, as it is a straightforward inductive proof on the argument monotype.

Finally, we can use this proof to define the following smart constructor for our Dynamic type as follows:

```
toDyn : (u : U n) → elV (FORALL u) → {p : isSoStrong u} → Dynamic
toDyn u x {p} = Dyn u (soundness u p) x
```

On the surface, it may seem like we have not accomplished much. After all, even this smart constructor requires a proof argument. There is, however, something, an important difference compared to the original constructor of the Dynamic type. This is best illustrated with an example.

We can now use the toDyn function to package the polymorphic identity function as a dynamic value:

```
idDyn : Dynamic
idDyn = toDyn u (λ x → x)
  where
  u : U (Succ Zero)
  u = (VAR Fz) ⇒ (VAR Fz)
```

What happened to the required proof argument? According to the type of toDyn we must also produce a proof of isSoStrong u. This proof, however, is *by definition* equivalent to a pair of unit types. That is the type isSoStrong ((Var Fz) ⇒ (Var Fz)) reduces to Pair Unit Unit. To complete the definition, we could pass (unit, unit) as the implicit argument to the toDyn function. As Agda supports η-expansion on record types, including the pair and unit type, the type checker is happy to fill in the missing implicit argument for us. This is where we can finally reap the rewards of our isSoStrong predicate: the *computation* reduces the required proof argument to a triviality that Agda is happy to infer. This limited form of proof automation through reduction to unit types is quite common in Agda developments (Swierstra, 2010; van der Walt & Swierstra, 2012).

Note that if we had defined the following erroneous version of idDyn, Agda would give an error message stating that it cannot find an argument of type Pair Unit (Pair Empty Unit) to pass to the toDyn

```
idDyn : Dynamic
idDyn = toDyn u (λ x → x)
  where
  u : U 2
  u = (VAR Fz) ⇒ (VAR Fz)
```

Examples. In this final section, we demonstrate several short examples of the cast function in action. Our first example shows that indeed, we can cast the polymorphic identity function to the monomorphic identity on natural numbers,

```
example₁ = cast u idDyn
  where
  u : U Zero
  u = NAT ⇒ NAT

test₁ : example₁ ≡ Just (λ x → x)
test₁ = Refl
```

More interestingly, we can also cast the polymorphic identity function to a polymorphic identity function on *pairs*. Doing so requires a shift from monotypes with a single free variable to monotypes with two free variables,

```
example₂ = cast u idDyn
  where
  u : U 2
  u = PAIR (VAR Fz) (VAR (Fs Fz)) ⇒ PAIR (VAR Fz) (VAR (Fs Fz))
test₂ : example₂ ≡ Just (λ x → x)
test₂ = Refl
```

Finally, we can also *reorder* quantified variables. The example below shows how to cast the const function of type **forall** a b . a → b → a to a function of type **forall** a b . b → a → b. To do so, use the constDyn dynamic defined previously:

```
example₃ = cast u constDyn
  where
  u : U 2
  u = (VAR (Fs Fz)) ⇒ ((VAR Fz) ⇒ (VAR (Fs Fz)))
test₃ : example₃ ≡ Just (λ x y → x)
test₃ = Refl
```

All these examples illustrate just how smoothly the cast function can handle the usual issues involved with variable binding, substitution, and unification.

6 Discussion

This paper is loosely based on a previous paper presented at the Workshop on Generic Programming (Van Noort *et al.*, 2011). The previous version was incomplete as two lemmas were postulated, but not proven. Also, it used a slight variation on McBride's unification algorithm (McBride, 2003), rather than implement the check function directly. The direct implementation presented here is much simpler and was originally presented in Van Noort's thesis (2012). The presentation there has been simplified further by introducing the requirement that the types stored in a dynamic may not contain spurious quantifiers.

6.1 Further work

Throughout this paper we have chosen a small universe with natural numbers that is closed under pairs and functions. It should be straightforward to add new types and type constructors, such as booleans or co-products. One obvious direction for further work is to stretch this universe further.

A limitation of the library presented here is that it can only handle predicative polymorphism. By using 'unsafe' Agda flags, such as disabling the termination checker or assuming Set : Set, we may be able to lift this restriction. This is a fairly high price to pay. One of the great advantages of the current implementation is the fact that it does not use any language extensions or postulates. By sticking to the safe fragment of Agda, our library has type soundness 'for free'.

A more feasible extension would be to drop the restriction that all quantifiers are in prenex form, but still disallow impredicativity. This would allow quantifiers to appear at within types, enabling us to write types such as **forall** a . a → **forall** b . b → a, which is not currently possible. This would require a change in our universes U and V that would certainly complicate matters.

More generally, this development illustrates how to write generic programs in a language with dependent types using an explicit universe construction. The drawback of such constructions is that they *only* work for a fixed universe. Better language support for such developments (Chapman *et al.*, 2010) would be very welcome. Until then, the best we can do is to parametrize our module with some universe, which the users of our library are free to instantiate. This does not work well for this development, however, as we define a large number of functions by induction on our universe, such as apply or check. Such functions must be passed as additional parameters to the module, together with any properties on which the development relies, thereby substantially increasing the burden on users.

An alternative direction for further work is to extend these ideas to handle *dependent types*, rather than the limited type quantification we have seen so far. Formalizing a dependently typed lambda calculi in type theory, however, is a notoriously hard problem (Barras, 1999; Danielsson, 2006; Chapman, 2008; McBride, 2010).

6.2 Related work

There is a great deal of literature comparing static and dynamic typing, and more specifically, discussing how to embed dynamic types safely in a language such as Haskell. Abadi *et al.* (1991) provide one of the first studies of how to incorporate dynamic typing in a statically typed language. While this initial work was restricted to monomorphic types, this was later extended to handle polymorphism (Abadi *et al.*, 1994). At the same time, Leroy and Mauny (1993) studied how to add polymorphic dynamics to ML.

Existing literature for dynamic typing in Haskell cannot handle polymorphism. Baars and Swierstra (2002) state: *Whether our approach can easily be extended with dynamic polymorphism is as yet unknown and a subject of further research.* In a related paper, Cheney and Hinze (2002) make a very similar observation: *We believe our Dynamic also can support making values of closed polymorphic types dynamic, although we have yet to experiment with unifying and pattern-matching polymorphic type representations.* A weaker research question has been formulated by Sheard *et al.* (2005) and said to be difficult (Sheard & Pasălić, 2008): *Is it possible to build [..] singleton types to represent polymorphic types? While we have tried many approaches we are not yet satisfied with the generality of any of them.* Unfortunately, there are no definitive answers to these questions.

How hard would it be to backport this development to Haskell? By using GADTs, type families, rank-n types, and other Haskell 98 extensions, Holdermans (2012) has already managed to develop a library along these lines. In contrast to the library presented here, however, every polymorphic dynamic must be instantiated to a monomorphic type by the cast function. Nonetheless, we would certainly hope that, in time, much of this work can be transferred to Haskell.

Acknowledgments

We would like to thank our colleagues in Nijmegen and Utrecht for their encouragement and suggestions. James McKinna was an excellent source of advice and entertaining discussions when we first embarked on this research. Bastiaan Heeren, Stefan Holdermans, Ruud Koot, José Pedro Magalhães, Stephanie Weirich and the anonymous reviewers provided invaluable feedback. We hope to have done justice with their comments.

References

Abadi, M., Cardelli, L., Pierce, B. & Plotkin, G. (1991) Dynamic typing in a statically typed language. *ACM Trans. Prog. Lang. Syst.* **13**(2), 237–268.

Abadi, M., Cardelli, L., Pierce, B., Rémy, D. & Taylor, R. (1994) Dynamic typing in polymorphic languages. *J. Funct. Prog.* **5**(1), 81–110.

Altenkirch, T. & McBride, C. (2003) Generic programming within dependently typed programming. *Proceedings of the IFIP TC2 Working Conference on Generic Programming*, Schloss Dagstuhl, Germany, July 2002.

Baars, A. & Swierstra, D. (2002) Typing dynamic typing. *Proceedings of the International Conference on Functional Programming (ICFP '02)*. Pittsburgh, PA, USA.

Barras, B. (1999, Nov) *Auto-Validation d'un Système de Preuves avec Familles Inductives*, Thèse de doctorat, Université Paris 7.

Brady, E. (2011) Epic – a library for generating compilers. In *Proceedings of the 12th International Conference on Trends in Functional Programming (TFP '11)*. New York: Springer-Verlag.

Chapman, J. (2008) *Type Checking and Normalisation*, PhD thesis, University of Nottingham, Nottingham, UK.

Chapman, J., Dagand, P.-É., McBride, C. & Morris, P. (2010) The gentle art of levitation. *Proceedings of the 15th ACM Sigplan International Conference on Functional Programming (ICFP '10)*, Baltimore, MD, September 27–29, 2010.

Cheney, J. & Hinze, R. (2002) A lightweight implementation of generics and dynamics. *Proceedings of the Haskell Workshop (Haskell '02)*, Pittsburgh, PA, USA.

Danielsson, N. A. (2006) A formalisation of a dependently typed language as an inductive-recursive family. *Proceedings of the Types for Proofs and Programs Conference(TYPES '06).*

Holdermans, S. (in preparation) *Polymorphic Dynamics for the Masses.*

Lämmel, R. & Peyton Jones, S. (2003) Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the Workshop on Types in Language Design and Implementation (TLDI '03)*, New Orleans, LA, pp. 26–37.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D. & Vouillon, J. (2011) *The OCaml System Release 3.12: Documentation and User's Manual*. Tech. Report, Institut National de Recherche en Informatique et en Automatique.

Leroy, X. & Mauny, M. (1993) Dynamics in ML. *J. Funct. Prog.* **3**(4), 431–463.

Martin-Löf, P. (1984) *Intuitionistic Type Theory*. Berkeley, CA: Bibliopolis.

McBride, C. (2003) First-order unification by structural recursion. *J. Funct. Prog.* **13**(6), 1061–1075.

McBride, C. (2010) Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming (WGP '10).*

McBride, C. & McKinna, J. (2004) The view from the left. *J. Funct. Prog* **14**(1), 69–111.

McBride, C. & Paterson, R. (2008) Applicative programming with effects. *J. Funct. Prog.* **18**(1), 1–13.

Norell, U. (2007) *Towards a Practical Programming Language Based on Dependent Type tTheory*, PhD thesis, Chalmers University of Technology, Sweden.

Norell, U. (2008) Dependently typed programming in Agda. In *Revised Lectures of the International School on Advanced Functional Programming, Heijen, The Netherlands*, Lecture Notes in Computer Science, vol. 5832, Koopman, Pieter, Plasmeijer, Rinus & Swierstra, Doaitse (eds). New York, NY: Springer-Verlag. pp. 230–266.

Oury, N. & Swierstra, W. (2008) The power of Pi. *Proceedings of the International Conference on Functional Programming (ICFP '08)*, Victoria, BC, Canada.

Peyton Jones, S. (ed). (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge, UK: Cambridge University Press.

Sheard, T., Hook, J. & Linger, N. (2005) *GADTs + Extensible Kinds = Dependent Programming*. Techical Report, Portland State University, Portland, OR.

Sheard, T. & Pasălić, E. (2008) Meta-programming with built-in type equality. *Electron. Notes Theor. Comput. Sci.* **199**, 49–65.

Swierstra, W. (2010) More dependent types for distributed arrays. *Higher Order Symb. Comput.* **23**(4), 489–506.

van der Walt, P & Swierstra, W. (2012) Engineering proof by reflection in Agda. In *24th International Symposium on Implementation and Application of Functional Languages, Oxford, UK, Revised Selected Papers(IFL '12)*. Oxford, UK: Department of Computer Science, University of Oxford.

van Noort, T. (2012) *Dynamic Typing in Type-Driven Programming*, PhD thesis, Radboud University Nijmegen, Netherlands.